ON PROXIMITY PROBLEMS

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

BODHISATTWA MUKHERJEE

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

JANUARY, 1988

CERTIFICATE

25/1/88.

This is to certify that the thesis entitled **"ON PROXIMITY PROBLEMS"** is a report of work carried out under my supervision by Bodhisattwa Mukherjee and that has not been submitted elsewhere for a degree.

, 25

Dr.A. Mukhopadhyay
Asst. Professor
Dept. of C.S.E.
I.I.T. , Kanpur.

Place: Kanpur

Date : January 1988.

ACKNOWLEDGEMENTS

It gives me great pleasure to put on record my feeling of indebtedness to Dr. A. Mukhopadhyay , my thesis supervisor , for creating an atmosphere of freedom in which it was a pleasure to work. Inspite of his busy schedule he spared enough time to give me regular guidance and ensure smooth progress.

I am also grateful to all my classmates and residents of Hall IV, IIT Kanpur for making my stay in this campus a memorable one.

B. Mukherjee.

BODHISATTWA MUKHERJEE

CONTENTS

Chapter		Page
1	INTRODUCTION	1
	1 Proximity problems	2
	1.1 Basic proximity problems	2
	1.2 The Voronoi diagram	5
	2 Organisation of the thesis	6
	3 References	7
2	A NEW ALGORITHM FOR THE CLOSEST PAIR PROBLEM	9
	1 Introduction	9
	2 Lower bound	10
	3 Sweep-rectangle technique	10
	4 One dimensional case	12
	5 Two dimensional case	15
	6 Generalisation to higher dimensions	21
	7 Conclusions	22
	8 References	23
3	A STUDY ON THE KTH DEGREE VORONOI DIAGRAM	25
	1 Introduction	25
	2 Definition of a Kth degree Voronoi diagram	27
	3 Properties of a Kth degree Voronoi diagram	28
	4 Construction of a Kth degree Voronoi diagram	29
	4.1 The Algorithm	30
	4.2 Analysis	33

5 Dynamic updation of a Kth degree Voronoi diagram 34
5.1 Data Structure 35
5.2 Insertion 37
5.3 Deletion 38
6 An incremental algorithm for the Kth degree Voronoi diagram 40
7 Discussions 41
8 References 42

4 SHORTEST PATH PROBLEM INSIDE A SIMPLE POLYGON IN THE PRESENCE OF OBSTACLES 44
1 Introduction 44
2 Algorithm to compute the shortest path tree of a simple polygon with obstacles inside 46
2.1 Convex polygon containing a single line-obstacle 47
2.2 Simple polygon containing K line obstacles 50
2.3 Convex polygon containing a single polygonal obstacle 60
3 Discussions 61
4 References 62

5 CONCLUSION 64

CHAPTER 1

INTRODUCTION

Computational Geometry , as it stands today , is concerned with the computational complexity of geometric problems within the framework of analysis of algorithms. A large number of applications areas such as pattern recognition , computer graphics , image processing , operations research , statistics , computer aided design , robotics , etc. , have been the incubation bed of the discipline since they provide inherently geometric problems for which efficient algorithms have to be developed. These problems include the Euclidean travelling salesman , minimum spanning tree , linear programming , and hosts of others. Algorithmic studies of these and other problems have appeared in the scientific literature with an increasing intensity in the past two decades and a growing number of researchers have been attracted to this discipline , christened "Computational Geometry" in a paper by Shamos [1] in 1975.

According to the nature of the geometric objects involved , we can identify basically five categories into which the entire collection of geometric problems can be conveniently classified , i.e. , convexity , intersection , geometric searching , proximity , and optimisation.

Since our research area is concentrated around the

proximity problem , we shall briefly describe the proximity problem and the work done in this area in the next few sections of this chapter,

1. PROXIMITY PROBLEMS

Geometric objects , such as points and circles , are used to model physical entities in the real world. In some cases we would like to have access to a suitable neighbourhood of the objects. For instance , in air traffic control we wish to keep track of the closest two aircrafts ; when aircrafts are modelled as points moving in space , we want to find the closest pair of points at a certain point in time. We shall first list a number of problems , some of which may appear unrelated , and describe a geometric construct , called a *Voronoi diagram* , which can be used to solve these problems within the same order of time spent for computing the diagram.

1.1. Basic Proximity problems

Problem 1 (Closest pair):

Given n points in the plane , find two points that are closest.

It is obvious that the generalisation of the problem in k dimensions , k >= 1 , can be solved in $O(k\ n^2)$ time by computing all interpoint distances. In one dimension , we can solve the problem easily in O(n log n) time by a preliminary sorting. It turns out that sorting does not generalise to higher dimensions. Using the divide-and-conquer

technique , Bentley and Shamos [2] showed that $O(n \log n)$ time is sufficient to solve this problem in dimensions $k >= 1$, and the time bound is optimal. An average case study of this problem is presented in [3] where an optimal average case algorithm is described.

Problem 2 (All nearest neighbours):

Given n points in the plane , find for each point a nearest neighbour (other than itself).

Problem 3 (Euclidean minimum spanning tree , EMST):

Given n points in the plane , find a tree that interconnects all the points with minimum total edge length.

This problem has an obvious application in computer networking where we want to interconnect all the computers at minimum cost. This formulation , however , forbids the addition of extra points. If additional points , called Steiner points , are allowed , the problem becomes the *minimal Steiner tree problem* , which has been shown to be NP-hard [4]. Note also that the EMST problem can be cast as a graph-theoretical problem , in which the weight of each edge is the distance between the two terminal vertices of the edge. In [5] several spanning tree algorithms have been illustrated. In general , the minimum spanning tree problem for a graph with n vertices requires $\Omega(n^2)$ time , for the minimum weight edge must be in the tree and there are $O(n^2)$ independent weights in the input ; however , the Euclidean metric properties can be exploited so as to solve the EMST

problem in O(n log n) time.

Problem 4 (Triangulation):

Given n points in the plane , construct a planar graph on the set of points such that each face within their convex hull is a triangle.

This problem arises in numerical interpolation of bivariate data where the function values are known at irregularly spaced points , and in the finite element method. A triangulation of these n points can be used to approximate the function value at a new point as the interpolation of the function values at the vertices of the triangle containing the new point.

Problem 5 (Nearest neighbour search) :

Given n points in the plane , with preprocessing allowed , find the nearest neighbour of a querry point.

This problem , also known as the "post office problem", arises in pattern classification [6] where the nearest neighbour decision rule is used to classify a new sample into the class to which its nearest neighbour belongs to , and in information retrieval where the record that best matches the querry record is retrieved [7].

Problem 6 (k Nearest neighbours search) :

The same as Problem 5 except that the k nearest neighbours are sought.

1.2. THE VORONOI DIAGRAM

The above problems can be solved efficiently by using Voronoi diagram. Given a set S of n points $\{p_1, p_2, \dots, p_n\}$, the *Voronoi diagram* [8] of S , denoted by Vor (S) , partitions the plane into n "equivalence" classes , each of which corresponds to a point. Specifically , the equivalence class corresponding to p_i is the Voronoi polygon $V(p_i)$, which is formally defined as $V(p_i) = \{r \mid r \text{ in } R^2$ and $d(r, p_i) <= d(r, p_j)$, $j <> i\}$. In other words , $V(p_i)$ is the locus of points that are as close to p_i as any other point of S and can also be defined as the intersection of the half planes $\bigcap_{i \neq j} H(p_i, p_j)$ where $H(p_i, p_j)$ is the half plane determined by the perpendicular bisector of $p_i p_j$ and containing p_i . Thus , the Voronoi diagram of a set of n points is just a collection of n Voronoi (convex) polygons , one for each point. The diagram is also called *Thiessen polygons* [9]. The properties of Voronoi diagram have been discussed in details in [10] , [11]. The straight-line dual of the Voronoi diagram is a triangulation of S. The triangulation is also known as *Delaunay triangulation* and *Dirichlet tesselation* [12]. The Voronoi diagram of a set S of n points in the plane can be constructed in O(n log n) time , which is optimal.

Once the Voronoi diagram is available , the closest pair problem , the all-nearest-neighbour problem , and the triangulation problem can all be solved in O(n) time

Obtain the straight line dual graph by scanning each edge of the Voronoi diagram ; since the dual graph is a triangulation and the total number of edges in Vor (S) is O(n) , the process takes O(n) time. The closest pair is identified with an edge of the triangulation and , similarly , the nearest neighbour of each point is given by an edge of the triangulation ; therefore , both problems can be solved in O(n) time. It has been shown [11] that the EMST is a subgraph of the delaunay triangulation. So the EMST problem can also be solved in additional O(n) time using the algorithm of Cheriton and Tarjan [5]. As for the nearest neighbour search problem , all we need to do is to find the Voronoi polygon in which the new point lies. The search is therefore a *point location problem* [13] , [14] and can be carried out in O(log n) time.

2. ORGANISATION OF THE THESIS

In this thesis we have looked at various kinds of *proximity problems.* In the second chapter of this thesis , a new O(n log n) algorithm is proposed for the closest-pair problem. This algorithm is based on a novel variation of the Sweep-line paradigm. Here we use a pair of lines whose distance from each other varies dynamically to sweep the point set.

In the third chapter a type of generalisation of Voronoi diagram called Kth degree Voronoi diagram is discussed. A new dynamising algorithm is given whereby a Kth degree Voronoi diagram can be updated efficiently. An easy-to-

implement incremental algorithm for constructing a Kth degree Voronoi diagram is also proposed in this chapter.

In the fourth chapter we have dealt with the shortest-path problem inside a simple polygon containing a constant K number of obstacles.

Finally , in chapter five we have concluded this thesis , mentioning some related open problems.

3. REFERENCES

[1] M.I.Shamos , "Geometric complexity", in Proc. 7th ACM Annu. Symp. Theory Comput., May 1975 , pp 224-233.

[2] J.l. Bentley and M.I. Shamos , "Divide-and Conquer in multidimensional space" , in Proc. 8th ACM Annu. Symp. Theory Comput. , 1976 , pp 220 - 230.

[3] J.L. Bentley , B. Weide , and A.C. Yao , "Optimal expected time algorithms for closest-point problems" , ACM Trans. Math. Software , vol. 6 , no. 4 , pp 563-579 , 1980.

[4] M. Garey , R.L. Graham , and D.S. Johnson , "Some NP-complete problems" , in Proc. 8th ACM Annu. Symp. Theory Comput. , May 1976 , pp 10-22.

[5] D. Cheriton and R.E. Tarjan , "Finding minimum spanning trees" , SIAM J. Comput. , pp 724-742 , Dec. 1976.

[6] R.O. Duda and P.E. Hart , Pattern classification and Scene Analysis. New York : Wiley , 1973.

[7] J.H. Friedman , J.L. Bentley , and R.A. Finkel , "An

algorithm for finding best match in logarithmic expected time " , ACM Trans. Math. Software , vol. 3 , no. 3 , pp 209-226 , 1977.

[8] C.A. Roger , Packing and covering. , Cambridge , England : Cambridge University Press , 1964.

[9] K.E. Brassel and D.E. Reif , "A procedure to generate Thiessen polygons" , Geogr. Anal. , vol. 11 , pp 289-303 , 1979.

[10] D.T. Lee , "Two dimensional Voronoi diagram in the L_p -metric" , J. ACM , Oct. 1980 , pp 604-618.

[11] M.I. Shamos and D. Hoey , "Closest pair problems" , in Proc. 16th IEEE Annual Symp. Found. Comput. Sci. , Oct. 1975 , pp 151-162.

[12] J. Bowyer , "Computing Dirichlet tesselations" , Comput J. , vol. 24 , pp 162-166 , 1981.

[13] F.P. Preparata , "A new approach to planar point location" , SIAM J. Comput. , vol. 10 , no. 3 , pp 473-482 , Aug. 1981.

[14] D.G. Kirkpatrick , "Optimal search in planar subdivisions" , SIAM J. Comput. , vol. 12 , no. 1 , pp 28-35 , Feb. 1983.

CHAPTER 2

A NEW ALGORITHM FOR THE CLOSEST PAIR PROBLEM

1. INTRODUCTION

The CLOSEST PAIR problem in computational geometry is the following :

> *Given a set of n points in the plane find a mutually closest pair.*

Apart from its intrinsic interest , an efficient solution to this problem would be useful , for instance , in air-traffic control.

Worst-case time optimal O(nlogn) algorithms for this problem have been given based on (a) the Divide-and-Conquer approach , (b) the construction of the Voronoi diagram of the given point set [5].

In this paper , we describe a new algorithm for the problem , based on a novel variation of the well known sweep-line paradigm in computational geometry.

The rest of the paper is organised as follows : In Section 2 we discuss a lower bound for the problem in the alge-

braic decision tree model. Section 3 contains a brief description of the sweep-line paradigm , and a detailed discussion of our variation of it , which we call the sweep-rectangle method. To motivate our solution to the problem , in Section 4 we discuss the case where all the points lie on a straight line. The general case is discussed in Section 5 and in the next section we discuss the case of d-dimensions. We summarise in Section 7 and indicate future directions of research.

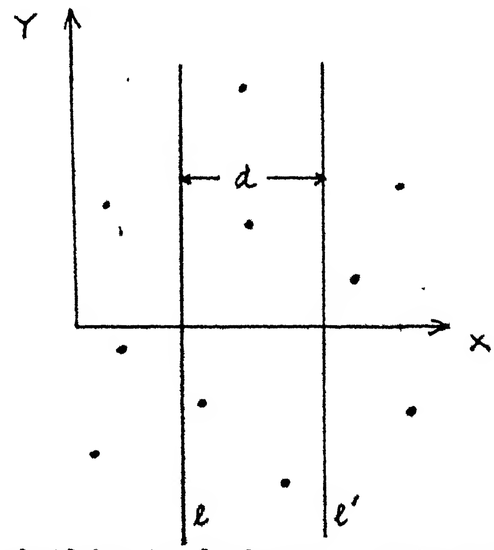
2. LOWER BOUND

The ELEMENT UNIQUENESS problem is to decide if n given real numbers are all distinct. We can transform this problem in linear time to the closest pair problem by considering the set of real numbers {x1 , x2 , ... , xn} as n points on the x-axis. Clearly , the elements are distinct if the distance between a closest pair is non-zero. Since in the algebraic decision tree model [1] [5] any algorithm that determines whether the elements of a set of n real numbers are distinct requires Ω (nlogn) tests , a lower bound of Ω (nlogn) is established for the CLOSEST PAIR problem.

3. SWEEP-RECTANGLE TECHNIQUE

The sweep-line paradigm is best understood by means of an example to which it has been successfully applied. To report all intersecting pairs among a set of n line segments in the plane , we sweep the set from -(inf) in the left to +(inf) in the right by a straight line moving in a direction

orthogonal to itself. The computation is supported by two dynamic data structures : The sweep-line status and the event point schedule. The former keeps record of the set of line segments which currently intersect the sweep-line and the latter maintains a list of points called event points at which all computations (like testing for intersection , deletion & insertion of line segments etc.) are done [4] [5] [6].



In our variant of this technique , two parallel lines l and l' , perpendicular to the x-axis , and at a distance d from each other which varies dynamically , is swept across the point set from left to right as shown in the above figure. (The two lines are the two sides of an infinitely long strip , and hence the name sweep-rectangle). Instead of the sweep-line status , here we have the sweep-rectangle status , which maintains the points currently inside the sweep-rectangle. The event point schedule is a lexicographically sorted queue of the given points. The sweep-rectangle status is updated at the event points. The update involves insertion of a new event point in the sweep-rectangle status

and deletion of the points going outside it. This update over , distances are computed between the newly inserted point and its nearest neighbour candidates inside the sweep-rectangle. A shortest distance pair among these is used to check if a closest pair of the points seen so far needs updating.

The width d of the rectangle decreases as it sweeps the point set. Initially it is chosen to be infinite and gets updated as the sweep goes on. At any moment of time , d gives the distance between a closest pair among the points seen so far. The elegance of this technique lies in the fact that the number of distance computations for each newly inserted point is constant. We can show that when a new point is inserted only the distances between the new point and a constant number of existing points are to be computed (discussed in details below). These constant number of points for each newly inserted point are called as the "nearest neighbour candidates" for that new point.

4. One dimensional case

Let us first try to develop an algorithm for the one dimensional case , using the above technique. W.l.o.g. we may assume that all the points lie on the x-axis. In this case the sweep-rectangle degenerates to an interval on the x-axis. It sweeps the point set from -(inf) to +(inf). A formal algorithm for this case is given below.

```
Procedure Sweep-rect-one-dim;
{It finds a closest pair (p,q) of a set of N points
 lying on the x-axis}
   begin
      Sort the N points and place them in the
      queue E;
      S := ϕ   ; /* S is the rectangle status */
      d := (inf);
      If the no.of points in E is more than one
         begin
            Extract the first two points p and q from E , find
            the distance between them and assign it to d;
            S := S ∪ {p,q} ;
            while (E <> ϕ ) do
                  begin
                        Extract from E the first element p1
                        Insert p1 in S;
                        Delete from S all the points having
                        abscissae less than p1 - d;
                        Compute the distance of p1 from the
                        other points in S;
                        Find the minimum of the above
                        distances (say d1);
                        Let q1 be a point at a distance
                        d1 from p1;
                        d := min (d1 , d);
                        If d = d1 then /* update p & q */
```

```
                        begin
                            p := p1;
                            q := q1;
                        end;
                end;
        end;
        Output d & (p , q);
    end;
```

Data structures

The data structure implementing the event point schedule has to support only the operations MIN (E) , which determines the smallest element in E and deletes it. We can use a linear list implementation of E which supports the above operation in O(1) time.

The data structure for the sweep-rectangle status is determined by the following lemma:

Lemma1: At any point of time , the rectangle contains at most two points.

Proof: The proof of the above lemma follows from the observation that d is the distance between a closest pair of the points met so far.

So , we can use a list having two elements to implement the sweep-rectangle status. This supports insert and delete operations in O(1) time.

Theorem1: The above algorithm correctly finds a mutually closest pair for a one dimensional distribution of points.

Proof: The proof follows from the fact that at any moment of time , d gives the distance between a closest pair and (p,q) gives the two points forming a closest pair for all the points to the left of the current event point.

Theorem2: The above one dimensional closest pair algorithm takes O(nlogn) time and uses O(n) space.

Proof: There are O(n) points in the point set and by Lemma1 , for each newly inserted point , we have to make O(1) distance computations. O(n) insertions take O(nlogn) time and O(n) deletions take O(n) time. The initial sorting step takes O(nlogn) time. So , the overall time complexity is O(nlogn).

The event point schedule uses O(n) space and the sweep-rectangle status takes O(1) space. So , the total space complexity is O(n).

5. Two dimensional case

Now , we extend the above one dimensional approach to two dimensions. Here, we have a vertical rectangle of varying width d sweeping the plane along the x-axis from -(inf) to +(inf). A formal algorithm for this case is given below.

```
Procedure Sweep-rect-two-dim;
{It finds a closest pair (p,q) of a set of n points in the plane}
  begin
    Sort the n points lexicographically first by
    x-coordinate and then by y-coordinate and
    place them in the queue E;
    S := Φ ; /* S is the rectangle status */
    d := (inf);
    If the no. of points in E is more than one do
      begin
        Extract from E the first two elements , find
        the distance between them and assign it to d;
        S := S ∪ {p,q} ;
        while (E <> Φ ) do
            begin
              Extract from E the first element p1
              Insert p1 in S;
              Delete from S all the points having
              abscissae less than (abscissae of p1 - d);
              Compute the distance of p1 from its nearest
              neighbour candidates in S;
              Find the minimum of the above distances
              (say d1);
              Let q1 be a point at a distance d1
              from p1;
              d := min (d1 , d);
              If d = d1 then    /* update p & q */
```

```
                    begin
                        p := p1;
                        q := q1;
                    end;
              end;
      end;
      Output d & (p , q);
  end;
```

Data structures

As in the one dimensional case , here also we can use a linear list to implement the event point queue. But the data structure for the sweep-rectangle status becomes slightly complicated.

The points in the sweep-rectangle status are kept sorted by y-coordinate (the reason will be clear latter). We can use a height balanced tree for storing the points ,sorted by y-coordinate. An extra link is added to every node of the tree. We call this x-link. The x-link of every node of the tree is manipulated in such a way that if we traverse the tree , using this extra link , we get the same points sorted by their x-coordinates.

The structure of a node can be implemented as follows:

```
type
     ptr = ^node;
     node = record
                 x       : co_ord;
                 y       : co_ord;
                 ylchild : ptr;
                 yrchild : ptr;
                 x-link  : ptr;
            end;
```

Since the event points to be inserted in the data structure comes sorted by their x-coordinates , the insert procedure is quite straightforward and can be written as :

```
Procedure INSERT(p);
   begin
      1. Insert the new event point p in
         the balanced tree , sorted
         by y-coordinates.
      2. x-link of the last of the
         previously inserted nodes is
         set to point to the currently
         inserted node.
   end;
```

Lemma2: The above INSERT procedure takes O(logn) time for

each point.

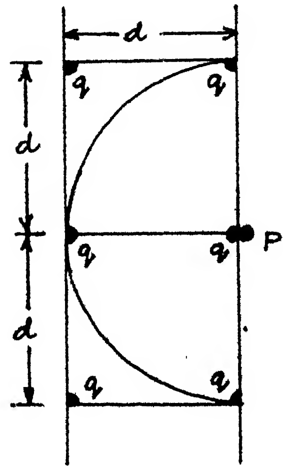
Proof: Step1 of the above INSERT procedure takes O(logn) time because insertion in a balanced tree takes O(logn) time. Step2 of the procedure takes O(1) time , since it does nothing except set a pointer to a node. So, the overall time complexity is O(logn).

The nodes are traversed through the x-link to execute the deletions of the points from the rectangle status. The delete operation for the rectangle status ia also straightforward and is as below :

```
Procedure DELETE;
    begin
        t := START;
        while t^.x < (p1.x - d) do
            begin
                delete the node;
                t := t^.x-link;
            end;
        store t^.xlink globally to be used as the
        START of delete for next delete operation;
    end;
```

Lemma3: The maximum number of distance computations required for any newly inserted point is at most 6.

Proof:



Let A and B be the vertical sides of the sweep-rectangle and p be the newly inserted event point. We must find all points q in the sweep-rectangle status that are within distance d of p. To do so , we note that the most dense packing of points inside the sweep-rectangle such that no two points are closer than d is as shown in the above figure. Thus p has at most 6 nearest neighbour candidates which means at most 6 new distances have to be computed.

Theorem3: The above algorithm correctly computes a mutually closest pair of the planar point set.

Proof: The proof follows from the observation that at any moment of time d and (p,q) gives the distance between a closest pair and a closest pair respec-

tively of the points seen so far.

Theorem4: The time complexity of the above algorithm for two dimensions is O(nlogn) and the space complexity is O(n).

Proof: To find the nearest neighbour of p we locate the node corresponding to p in the sweep-rectangle status. This takes O(logn) search time. Then find out the successors and predecessors of p (total no. of them is bounded above by 6) to get the nearest neighbours. This totally takes O(logn) time in the worst case. Thus the nearest neighbours of p can be found in O(logn) time. Since the total number of points in the plane is n , this step takes O(nlogn) time. Also insertions into the sweep-rectangle status take totally O(nlogn) time and deletions takes totally O(n) time in the worst case. The initial sorting step takes O(nlogn) time.

Therefore , the overall worst case time complexity of the above algorithm is O(nlogn).

The data structure for the event point schedule uses O(n) space and the sweep-rectangle status also uses O(n) space. So , the overall space complexity is O(n).

6. Generalisation to higher dimensions

For simplicity , instead of d-dimensions , let us gen-

eralise the algorithm and the data structures to three dimensions. Here instead of a rectangle with two vertical sides , two hyperplanes parallel to the yz plane , with varying distance of separation d , sweeps the three dimensional space along the x-axis. Initially the points are sorted lexicographically first by their x-values , then y-values and finally by their z-values , and placed in a queue E. The queue E can be implemented in the same way as in the one and two dimensional cases above.

Since the distance between any two points is >= d it is seen that here also the maximum number of nearest neighbour candidates for every newly inserted point is O(1). Hence , for each newly inserted point the number of distance computations is O(1). The rest of the algorithm and data structures are the same as for the two dimensional case.

But the optimality of the two dimensional algorithm does not carry over to higher dimensions. In higher dimensions , we don't find an appropriate data structure for the sweep-rectangle status which offers us a O(logn) insertion and a O(logn) deletion. Also in the higher dimension the simplicity , one of the major advantages of this sweep-rectangle technique vanishes.

7. CONCLUSIONS

We have outlined an optimal O(nlogn) algorithm for the CLOSEST PAIR problem for a planar point set , based on the novel concept of a sweep-rectangle. However , this optimal-

ity does not carry over to higher dimensions. But this does not detract from the practical usefulness of the algorithm since the number of dimensions <= 3 in practical situations. It would be interesting to investigate if the sweep-line or sweep-rectangle technique can be successfully applied to other problems for which optimal algorithms have been designed based on other paradigms like the Divide-and-Conquer , and vice versa. Guting , for example , has given optimal algorithms based on the Divide-and-Conquer strategy for various rectangle geometry problems for which optimal algorithms based on the sweep-line technique were given earlier [2] [3].

8. REFERENCES

[1] Ben-Or , M. , Lower Bounds for Algebraic Computation Trees. Proc. 15th ACM Annual Symp. on Theory of Comput. , pp 80 - 86 , May 1983.

[2] Guting , R. H. , Optimal Divide-and-Conquer to Compute Measure and Contour for a Set of Iso-Rectangles. , Acta Informatica , Vol 21 , pp 271 - 291 , 1984.

[3] Guting , R. H. , Conquering Contours: Efficient Algorithms for Computational Geometry , Report , der Universitat Dortmund an der Abteilung Informatik , 1983.

[4] Nievergelt , J. , and Preparata , F.P. , Plane-Sweep Algorithm for Intersecting Geometric Figures. CACM 25 , pp 739 - 747 , 1982.

[5] Preparata , F.P. , and Shamos, M.I. , Computational Geometry , An Introduction. Springer Verlag , 1985.

[6] Shamos , M.I. , and Hoey , D. , Geometric Intersection Problems. Seventeenth Annual IEEE Symposium on foundations of Computer Science , pp 208 - 219.

CHAPTER 3

A STUDY ON THE Kth DEGREE VORONOI DIAGRAM

1. INTRODUCTION

Given a set $S = \{q_1, q_2, \ldots, q_n\}$ of n points (called sites) in the 2-dimensional plane with each point q_i represented as an ordered pair (x_i, y_i), $i = 1, 2, \ldots, n$, let $d(q_i, q_j)$ denote the Euclidean distance between the two points q_i and q_j. The locus of points closer to q_i than q_j, denoted by $h(q_i, q_j)$, is one of the half planes determined by the bisector $B(q_i, q_j)$ and is $= \{r \mid d(q_i, r) < d(q_j, r)\}$. The locus of points closer to q_i than any other point in S, denoted by $\nu(i)$, is thus given by $\nu(i) = \bigcap_{i \neq j} h(q_i, q_j)$, the intersection of all the half planes associated with q_i. Vertices of the Voronoi polygon are called Voronoi points and their boundary edges are called Voronoi edges. The set of Voronoi polygons partitions the plane into n regions, some of which may be unbounded, and is referred to as the Voronoi diagram V(S) for the set S of n points. It has been shown that the Voronoi diagram for a set of n points in the plane can be constructed in O(nlogn) time.

There has been a number of extensions and generalisations of the Voronoi diagram. The definition of the Voronoi diagram given above can be easily extended to the L_p -metric where 1 <= p <= (inf) [1] , and the diagram can still be constructed in O(n log n) time , if we allow that the computation of the pth root can be done in constant time.

The second extension consists of considering the Voronoi diagram of a set of objects rather than points. In [2] the Voronoi diagram for a set of line segments or circles is considered and an $O(n \log^2 n)$ time algorithm is given for its construction. The time bound was later improved by Kirkpatrick to O(n log n) [3].

The third extension focuses on the fact that in the Voronoi diagram discussed so far , each polygon is the locus of points nearest to one point. To be more precise the diagram should be termed the *nearest neighbour Voronoi diagram.* Shamos and Hoey [4] considered the order-k Voronoi diagram of a set of points where each polygon of the diagram is associated with k points , k >= 1 , with the property that for any point inside the polygon its k nearest neighbours are precisely the associated k points. With the order-k diagram the k-nearest neighbours search problem can be solved in O(log n + k) time , where the first term accounts for point location and the second term for reporting the answer. Properties and a method for the construction of the order-k Voronoi diagram can be found in [5] , [6] , [4].

The fourth extension is to associate each point with a positive weight , resulting in a "weighted" Voronoi diagram [7]. The weighted Voronoi diagram consists of n "regions", each of which is the locus of points whose weighted distance to a given point is minimum. An $O(n^2)$ algorithm for constructing such a weighted diagram can be found in [8].

Another extension consists of generalising the diagram or triangulation to higher dimensions. Some results in this direction can be found in [8] , [10] , [11] , [12].

In this chapter we are going to discuss a generalisation of the Voronoi diagram , called the higher degree Voronoi diagram. In this higher Voronoi diagram , we partition the plane into polygons in such a way that every polygon gives the locus of the points kth closest to a particular site.

Kth degree Voronoi diagram , as it is called , has many practical applications. For example , in an information retrieval system , the request of searching for the kth nearest records to a query in a file with n records is quite common. And this can be solved using a kth degree Voronoi diagram.

2. DEFINITION OF A Kth DEGREE VORONOI DIAGRAM

For a given set $S = \{q_1 , q_2 , \ldots , q_n\}$ of n points in the 2-dimensional plane , a kth degree Voronoi diagram , denoted by $V^k (\{q_1 , \ldots , q_n\})$, or $V^k (S)$ for short , is a partition of the plane into some convex Voronoi polygons.

Each kth degree Voronoi polygon , denoted by V^k $(<q_1 , q_2 , \ldots , q_k>)$ is asociated with an ordered set of k points and is the locus of the points closest to the site q_1 , second closest to the site q_2 , ... , kth closest to q_k. For a particular site we define the corresponding kth degree Voronoi region as

$$V^k (q_i) = \{x \mid d (x , q_i) \text{ is the kth smallest of the sequence } \{d(x , q_i) \mid i = 1 , 2 , .. ,n\}\}$$

It is nothing but the union of a disjoint set of Voronoi polygons each of which is associated with an ordered set of k points , q_i being the kth in each. So , V^k (q_i) gives the locus of the points kth closest to the site q_i in the plane. V^k (q_i) may also be empty.

A kth degree Voronoi diagram can also be expressed in terms of higher order Voronoi diagrams. In fact ,

$$V^k (S) = V_k (S) \cap V_{k-1} (S) ,$$

where V_k (S) is the order-k Voronoi diagram of the given set S of n points.

3. <u>PROPERTIES OF A Kth DEGREE VORONOI DIAGRAM</u>

In this section , we talk about the general properties of a kth degree Voronoi diagram.

Prop1: Each polygon in a kth degree Voronoi diagram is associated with an ordered set of k points from the set S , and there exists at most one polygon corresponding to each ordered set.

Proof: The points inside a polygonal region of the kth degree Voronoi diagram is nearest to a point q_1 of the set S , second nearest to a point q_2 , third nearest to a point q_3 and so on. So , we get an ordered set (q_1 , q_2 , q_3 , ... , q_k) of cardinality k which identifies the polygon.

Prop2: In a kth degree Voronoi diagram V^k (S) , if q is a point on a Voronoi edge , then kth nearest neighbour of q is not unique.

Proof: Straight forward.

Prop3: Total number of regions in a kth degree Voronoi diagram is O(kk!(n-k)).

Proof: The proof comes directly from the fact that the total number of regions in the kth order Voronoi diagram is O(k(n-k)).

4. CONSTRUCTION OF A Kth DEGREE VORONOI DIAGRAM

Our algorithm for constructing a kth degree Voronoi diagram is very simple and follows the line of Lee [6] for the construction of a order-k Voronoi diagram. The algorithm is iterative , i.e. , initially we get V^1 (S), then get V^2 (S) , and so on until we get V^k (S) from V^{k-1} (S) for a specified k and a set S of n points. What happens in the course of iteration is that the Voronoi polygons of the previous degree Voronoi diagram get divided into subpolygons. The main difference of the algorithm for the kth order Voronoi diagram from that of the kth degree is that in the

former case , in the course of iteration , new points and edges are formed but in the process some old points and edges get deleted. But in the latter case new Voronoi points and edges are formed but no point or edge gets deleted. This fact adds to the complexity of our algorithm for constructing a kth degree Voronoi diagram.

4.1. THE ALGORITHM

In this section we shall give an algorithm to partition a single Voronoi polygon , say $V^1(\langle p_n \rangle)$, of a degree 1 Voronoi diagram $V^1(S)$ to get the subpolygons for $V^2(S)$. Suppose that the polygons $V^1(\langle p_0 \rangle)$, ... , $V^1(\langle p_{m-1} \rangle)$ are adjacent to polygon $V^1(\langle p_n \rangle)$. We want to partition $V^1(\langle p_n \rangle)$ into m subregions such that each subregion r_i is the locus of points closer to p_i than to any other points except p_n for i = o , 1 , ... , m-1. To partition $V^1(\langle p_n \rangle)$ thus , we have to compute its intersection with $V^1(\{p_0 , p_1 , \ldots , p_{m-1}\})$. The effect of this is the extension of the edges intersecting at the vertices of $V^1(\langle p_n \rangle)$ to the interior of $V^1(\langle p_n \rangle)$, thereby partitioning the polygon $V^1(\langle p_n \rangle)$. Let the vertices of $V^1(\langle p_n \rangle)$ be denoted as I_0 , I_1 , ... , I_{m-1}. Each edge (I_i , I_{i+1}) of the polygon is a portion of the bisector between p_i , p_n and is represented by the index pair (i , n). By assumption each vertex I_i is an intersection of three edges represented by (i , n) , (i-1 , n) , (i , i-1). Let us denote the edge which is incident with I_i and which is not on the boundary of the polygon $V^1(\langle p_n \rangle)$ by $IN(I_i)$. The following figure shows a typical Voronoi

polygon V^1 ($<p_{1\emptyset}>$) which is to be partitioned.

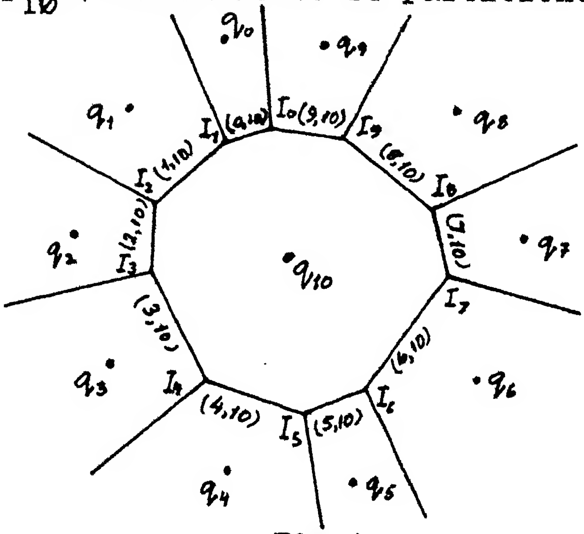


Fig 1.

We shall tackle the problem by divide - and - conquer technique. We first obtain the Voronoi diagram for sets of three points $\{p_{m-1}, p_\emptyset, p_1\}$; $\{p_2, p_3, p_4\}$ etc. by extending the edges associated with $\{ IN(I_\emptyset), IN(I_1) \}$, $\{IN(I_3), IN(I_4)\}$ etc. , respectively , into the interior of V^1 ($<p_n>$). By merging two adjacent Voronoi diagrams for sets of three points , we get the Voronoi diagram for a set of six points. Repeating this merge process $\lceil \log_2 m/3 \rceil$ times , we will obtain the Voronoi diagram for m points. The edges of the diagram which are interior to V^1 ($<p_n>$) will partition V^1 ($<p_n>$) into m subregions. Fig 2 shows the merge process of two Voronoi diagrams for $\{p_9, p_\emptyset, p_1\}$ and $\{p_2, p_3, p_4\}$ and Fig 3 shows the final merge process for the two Voronoi diagrams for $\{p_9, p_\emptyset, \ldots, p_4\}$ and $\{p_5, p_6, \ldots, p_9\}$. In Fig 3 , the merge process starts with

the extension of $IN(I_5)$ and ends at the point E as shown.

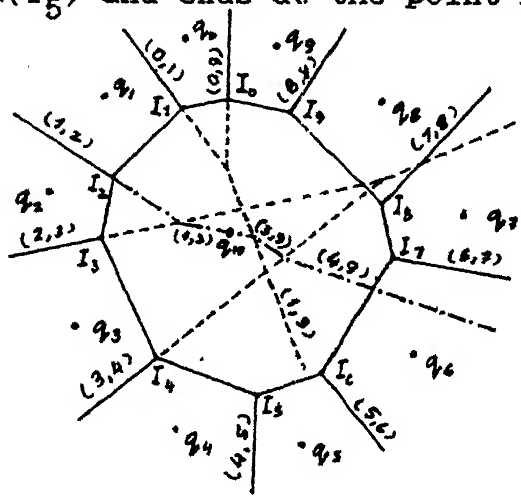


Fig 2.

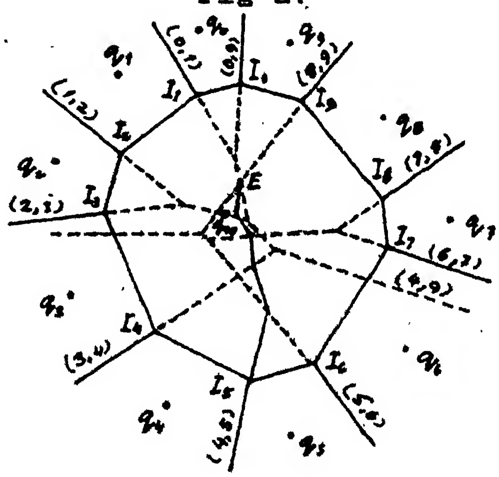


Fig 3.

The technique used to merge two Voronoi diagrams is discussed in details in [4]. Here we shall omit the details of the merge process and describe a method of identifying the set of Voronoi points on which the divide - and - conquer technique is to be applied.

Suppose that we have obtained a degree i Voronoi diagram $V^i(S)$, $k > i >= 1$. We can divide the Voronoi points into two groups -- old points which exist from previous degrees and new points which are just created. It is

easy to see that only the set of new Voronoi points are needed in order to construct the $V^{i+1}(S)$ diagram. Note that all the vertices in $V^1(S)$ are new Voronoi points. Now to partition $V^i(\langle p_1, p_2, \ldots, p_i \rangle)$ we first obtain the set of new Voronoi points. Such new Voronoi point I_j is associated with an edge $IN(I_j)$. Now to partition $V^i(\langle p_1, p_2, \ldots, p_i \rangle)$ we first obtain the set of new Voronoi points I_j and then apply the divide-and-conquer technique to the set as just described. After each Voronoi polygon is partitioned, we need to associate it with a new set of $(i + 1)$ points, and at the same time mark the new Voronoi points for use in the next iteration. In this manner we can obtain the $(i + 1)$th degree diagram $V^{i+1}(S)$.

4.2. ANALYSIS

Now let us analyse the running time of the algorithm. Suppose that the polygon $V^i(\langle p_1, p_2, \ldots, p_i \rangle)$ to be partitioned has s new Voronoi points $I_1, I_2, \ldots, I_s$. Since there are $O(i!n)$ Voronoi polygons in $V^i(S)$ and $O(i\,n)$ new Voronoi points [6], the total number of operations required is

$$\sum_{1}^{(i!n)} O(s_j \log s_j) = O(iN \log N)$$

Since $(k-1)$ iterations are required to obtain a kth degree Voronoi diagram, the worst case complexity for the entire work is :

$$\sum_{i=1}^{k-1} O(i\, n \log n) = O(k^2\, n \log n).$$

5. DYNAMIC UPDATION OF A Kth DEGREE VORONOI DIAGRAM

For on-line applications , dynamic updation of Voronoi diagrams is very important. Most of the real-life problems demand an on-line algorithm. Since a kth degree Voronoi diagram is a very complex data structure , dynamising it creates some difficulties. We are not aware of any efficient dynamising technique for constructing a kth degree Voronoi diagram. Here , we propose an easy-to-implement dynamising technique. But before we go into the details of the algorithm , we'll like to explore a few more properties of a kth degree Voronoi diagram.

Prop4: Each addition of a point divides an existing polygon in at most k parts.

Proof: The proof becomes very simple if we think each polygon as nothing but an ordered set of points. Let a polygon P in a kth degree Voronoi diagram be denoted by the ordered set $(q_1 , q_2 , \ldots , q_k)$, Now , the addition of a new point q_{nn} may divide this polygon into sub regions. Since the order of $(q_1 , q_2 , \ldots , q_k)$ remains the same for all of the subregions , they'll have these points in the same order in their associated ordered sets with q_{nn} inserted between any two consecutive points and with q_k deleted from the set to maintain the cardinality of the set. Since there are a maximum of k places where q_{nn} can be inserted , the property is established.

Prop5: Deletion of an existing point may cause adjacent regions to merge.

Proof: Straight forward.

Before we come to the formal algorithm , we'll briefly discuss the data structure used.

5.1. DATA STRUCTURE

We can use any of the standard data structures like DCEL to implement a kth degree Voronoi diagram. But with this Voronoi diagram , we have to store some additional information.

1.Ordered Set:

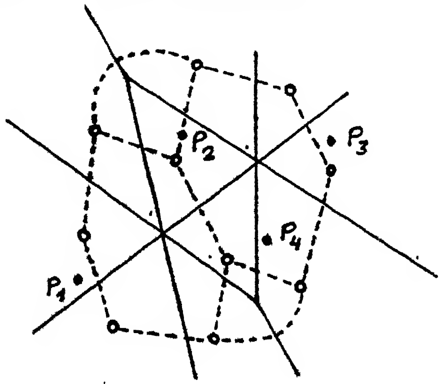
With every polygon , we have to store an ordered list containing the ordered set associated with that polygon

2.Dual:

We also have to store the dual of this kth degree Voronoi diagram separately which is to be updated with the kth degree Voronoi diagram. Initially we shall construct a spanning tree of the dual of the Voronoi diagram and store it along with the dual. We have to update the spanning tree when the dual is updated. Construction of the spanning tree from the dual initially will require $O(n^2)$ preprocessing time.

Def: We define the dual of a kth degree Voronoi diagram as

a graph where each point corresponds to one polygon of the kth degree Voronoi diagram and each edge between two points gives the adjacency of the corresponding polygons. For Example:



Second Degree Voronoi diagram For 4 Points.

— Voronoi diagram

-- dual

The dual of a kth degree Voronoi diagram has the following properties:

1. *The dual may have cycles in it.*

2. *The dual may be used to get the neighbours or adjacent regions of a given polygon.*

3. *A spanning tree of this dual can be used to traverse each of the polygons of a kth degree Voronoi diagram efficiently without repetition.*

Now , we are ready to present the formal algorithms for insertion of a new point in the Voronoi diagram and deletion of an existing point from it.

5.2. INSERTION

The following steps are taken when a new point q_{nn} is inserted into the existing Voronoi diagram.

```
Procedure Insert (q_nn : new_point) ;

   STEP1: Take the dual of the kth degree
          Voronoi diagram;
   STEP2: Traverse the Voronoi diagram using
          a spanning tree of it;
          FOR each polygon DO
            BEGIN
              Let the ordered set associated
              with the polygon be (q_1 , q_2 ,
              ... ,q_k);
              So the polygon is represented
              by V(<q_1 , q_2 , ... , q_k>);
              Take the bisectors between q_nn
              and q_1 , q_nn and q_2 , and so
              on and divide the polygon into
              O(k) parts;
              Form the ordered set for each
              new subpolygons by inserting
              q_nn in the ordered list (q_1 ,
              q_2 , ... , q_k) in appropriate
              position;
              Delete q_k from the ordered lists
              of the subpolygons , if necessary ,
              to maintain the cardinality of
```

the ordered set;

update dual and its spanning tree;

END.

ANALYSIS

There are O(k!n) polygons in the kth degree Vornoi diagram for a fixed k and S. Each polygon gets divided into O(k) parts. The manipulation of the existing spanning tree requires O(1) time. So the worst case time complexity of the above algorithm is O(k k! n)

5.3. DELETION

Deletion of an existing point is not as simple as the insertion and creates some major problems. We directly give the formal steps for the process of deleting an existing point from a kth degree Voronoi diagram.

Procedure Delete ;

{Let p_d be the point to be deleted}

STEP1: Traverse the Voronoi diagram using a spanning tree of the dual graph and delete the point p_d from all the ordered lists.

STEP2: Traverse the dual and perform the following steps:

STEP2.1: Find neighbours of a polygon using the adjacency property

of the dual.

STEP2.2: Check if the ordered set associated with any neighbouring polygon is same as that of the current polygon. If yes , coalesce them into a single polygon;

STEP3: There will be a number of regions having an ordered set of cardinality (k-1). To maintain the cardinality to k through out the Voronoi diagram , divide those polygons again according to the surroundings. Each of the sub-polygons generated will have an ordered set of cardinality k.

[DIVISION ALGORITHM HAS ALREADY BEEN PRESENTED IN SECTION 4]

STEP4: Update the existing dual and its spanning tree;

ANALYSIS

Since there are O(k! n) nodes totally , Step1 takes O(k! . n) time in the worst case. Let e be the number of edges in the dual graph. So the number of Voronoi edges also becomes equal to e. Since we traverse each edge of the dual graph at most two times , and since number of comparisons required in Step2.2 for each two ordered sets is O(k) ,

Step2 takes totally O(ke) time. Since the number of new points is O(k n) , total time required in Step3 is O(k n log n) in the worst case. Dynamic manipulation of the existing dual graph and its spanning tree requires O(n) time in the worst case. So the worst case time complexity of the above algorithm is O(max (k! n , ke , k n logn).

6. AN INCREMENTAL ALGORITHM FOR THE kth DEGREE VORONOI DIAGRAM

Incremental algorithms are very useful for online applications. Once we get the insertion algorithm for kth degree Voronoi diagram , designing an incremental algorithm becomes easy and straight forward. When we start constructing a kth degree Voronoi diagram incrementally for a fixed k , we have to cross through 2 phases.

PHASE1:

When the total number of the points currently present on the plane < k

PHASE2:

When the total number of points present >= k

Once we reach Phase 2 i.e. the total number points in the plane exceeds k , the incremental algorithm becomes same as the insertion algorithm discussed in the last section. But for Phase 1 , we have to design a separate algorithm. For constructing a kth degree Voronoi diagram , we first have to cross Phase 1 , then go into the Phase2. While inside Phase1 we take the following approach: If n be the

number of points at present on the plane then construct V^n (S) i.e. always we'll have a nth degree Voronoi diagram in our hand. So the incremental algorithm, in Phase 1, takes the following form:

Procedure Phase1 ;

STEP1: Locate the new point in V^n (S).

STEP2: The new point p_n divides each polygon atmost in n parts,

STEP3: For each polygon, take the bisectors between p_n and each of the members in its ordered set and use those bisectors to divide the polygon in O(n) parts.

STEP4: Update the dual and its spanning tree.

So after the above steps we get V^{n+1} (S') where S' = S U $\{p_n\}$. We repeat the above steps till n = k, and then we reach Phase2, the insertion algorithm.

The time complexity of the above algorithm in Phase1 is $O(\sum_{i=1}^{k} i^2 \; i!)$ So the complexity of the entire incremental algorithm can be calculated to be $O(\sum_{i=1}^{k} i^2 \; i! + (n-k)kk! \; n) = O(k^3 \; k! + (n-k)kk! \; n) = O(kk!(k + (n-k)n))$.

7. DISCUSSIONS

The algorithm presented here for constructing a kth degree Voronoi diagram is iterative and so conceptually easy. The strategy proposed here for the dynamic updation of

a kth degree Voronoi diagram is conceptually easy and also easy to implement. The same strategy can also be used for the order k Voronoi diagram. There is scope for furthur work in the following directions :

1. Sweep line algorithm for a kth degree Voronoi diagram.
2. Construction of kth degree Voronoi diagram in different metrics.
3. Weighted kth degree Voronoi diagram.
4. Geodesic kth degree Voronoi diagram

8. REFERENCES

[1] D.T.Lee , "Two dimensional Voronoi diagram in the L_p metric" , J.ACM. , Oct., 1980 , pp 604-618.

[2] D.T.Lee and R.L. Drysdale , "Generalised Voronoi diagram in the plane" , SIAM J. Comput. , vol. 10 , no. 1 , pp 73-87 , Feb. , 1981.

[3] D.G. Kirkpatrick , "Efficient computation of continuous skeletons" , in Proc. 20th IEEE Annu. Symp. Found. Comput. Sci. , Oct. , 1979 , pp 18-27.

[4] M.I. Shamos and D. Hoey , "Closest-points problems" , in Proc. 16th IEEE Annu. Symp. Found. Comput. Sci. , Oct. 1975 , pp 151-162.

[5] H. Edelsbrunner , J. O'Rourke , and R. Siedel , "Constructing arrangements of lines and hyperplanes with

applications" , in Proc. IEEE Annu. Symp. Found. Comput. Sci. , Nov. , 1983 , pp 83-91.

[6] D.T. Lee , "On k-nearest neighbour Voronoi diagrams in the plane" , IEEE Trans. Comput. , vol. C-31 , pp 478-487 , June 1982.

[7] B.N. Boots , "Weighting Thiessen polygons" , Econ. Geogr. , pp 248-259 , 1979.

[8] F. Aurenhammer and H. Edelsbrunner , "an optimal algorithm for constructing the weighted Voronoi diagrams in the plane" , Pattern Recognition. vol. 17 , no. 2 , pp 251-257 , 1984.

[9] J.D. Boissonnat and O.D. Faugeras , "Triangulation of 3D objects" , in Proc. 7th Int. Joint Conf. Artificial Intell. , Canada , 1981 , pp 658-660.

[10] W. Brostow , J.P. Dussault , B.L. Fox , "Construction of Voronoi polyhedra" , J.Comput. Phys. , vol. 29 , pp 81-92 , 1978.

[11] R. Siedel , "The complexity of Voronoi diagrams in higher dimensions" , in Proc. 20th Allerton Conf. Commun. Control and Comput. , 1982 , pp 94-95.

[12] D.F. Watson , "Computing the n-dimensional Delaunay tesselation with applications to Voronoi Polytopes" , Comput. J. , Vol. 24 , no. 2 , pp 167-172 , 1981.

CHAPTER 4

SHORTEST PATH PROBLEM INSIDE A SIMPLE POLYGON IN THE PRESENCE OF OBSTACLES

1. INTRODUCTION:

Recently there has been a significant upsurge of results concerning geometry inside a simple polygon which include an improved triangulation algorithm [1] , efficient algorithms for calculating the geodesic center and the geodesic diameter of a polygon , a number of new shortest-path and visibility-related algorithms that require linear amount of time beyond a triangulation [2] , and algorithms for link distance problems [3]. Some of this work concentrates on internal distance analogs of fundamental problems for point sets in the Euclidean plane. For example , Toussaint [4] developed an algorithm for computing the "relative convex hull" of a set of points , which is the shortest cycle containing all given points and contained in a given simple polygon.

Our present work extends the shortest-path problem inside a simple polygon to the case where the simple polygon

ontains some obstacles inside it. The problem can be formally stated as follows:

Given a fixed source point X inside a polygon P (convex or simple) containing a number of obstacles inside it, calculate the shortest paths inside P from X to all vertices of P and provide a preprocessing of P into a data structure from which the length of the shortest-path inside P from X to any desired target point Y can be found in time O(log n); the path itself can be found in time O(logn + k), where k is the number of segments along the path.

Some work has been done previously towards the design of an algorithm for such preprocessing of P but under the assumption that P does not contain any obstacles inside it [2].

Our algorithm uses the familiar sweep-line strategy to preprocess P. However, peculiarities of this problem necessitate a somewhat non-standard implementation of the strategy (it will be evident from the discussions in the later sections).

Possible applications of our algorithm include the closest point problem, the nearest post-office problem, the problem of finding the shortest-path to any target point from a particular source point in the context of a polygonal universe, such as an (polygonal) island with interior lakes or a polygonal factory floor with interior lawns etc. The

presence of obstacles inside the polygon makes the problem far more useful in real life-applications.

This chapter is organised as follows. Section 2.1 describes an algorithm for a convex polygon containing a single obstacle in the form of a straight line segment which we shall call a line-obstacle hereafter. Section 2.2 describes an algorithm for the case when P is a simple polygon and contains a constant K number of line-obstacles inside it , where K >= 1. Section 2.3 describes the algorithm for preprocessing a convex polygon containing a single polygonal obstacle. Lastly in Section 3 we conclude this chapter , mentioning the possible extension of the algorithm of Section 2.3 to the case when P is simple and number of polygonal obstacles inside it is more than one and some related open problems.

2. <u>ALGORITHM TO COMPUTE THE SHORTEST PATH TREE OF A SIMPLE POLYGON WITH OBSTACLES INSIDE</u>

Let P be a convex or simple polygon with n vertices and let s be the given source point on or inside P. For each vertex v of P let PA (s,v) denote the Euqlidean shortest-path from s to v lying inside P and |PA (s,v)| the length of this path. It is well known that $\leqq$ PA (s,v) taken over all vertices v of P , is a planar tree Q_{sp} (rooted at s) , which we call the *shortest-path tree* of P with respect to s. This tree has altogether n nodes , namely the vertices of P , and its edges , in this case , are ~~either~~ straight line segments connecting these nodes to the same point. Our goal

is to compute this tree and to partition P in such a way that the length of the shortest-path inside P from X to any desired target point Y can be found in O(log n) time.

Before we go to the case of a simple polygon with K polygonal obstacles inside, we would like to discuss some simpler cases. First we give an algorithm for preprocessing P when P is convex and contains a single line-obstacle. Then we propose a sweep-line strategy for the second case i.e. when P is simple polygon containing K line-obstacles. Then we describe an algorithm when P is convex and contains a single polygonal obstacle. Finally we discuss how this algorithm can be extended to the case when P is a simple polygon and contains more than one polygonal obstacles.

2.1. CONVEX POLYGON CONTAINING A SINGLE LINE-OBSTACLE

This case is very simple and we don't require the sweep-line strategy to solve this problem. It is easy to see that without any obstacle inside, PA (s,v) is nothing but a straight line from s to v. The presence of an obstacle divides the convex polygon into three sub-regions so that shortest-path to any point in a particular region passes through the same corner points.

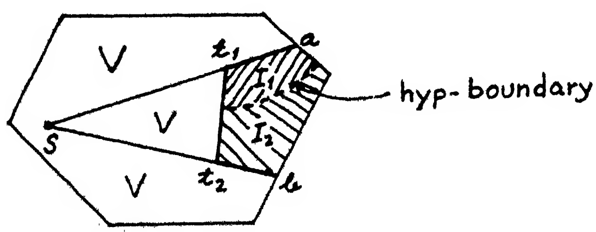


FIG. 1

In the above figure s is the source point and O is the line-obstacle inside the polygon P. The straight lines st_1 and st_2 are the tangents to the obstacle O drawn from the source vertex s. Let the straight lines st_1 and st_2 intersect the boundary of the convex polygon P at ponts a and b respectively. So we get two regions inside the polygon -- I , the region [t_1 t_2 b a] in P invisible from s and V , the remaining portion of P visible from s.

$$P = V \cup I$$

Now , PA (s,v) is a straight line segment joining s and v when $v \in V$. But PA (s,v) is a polygonal path whose corner is the point t_1 or t_2 when $v \in I$. For $v \in I$,

$$PA\ (s,v) = \min\ (st_1 \cup t_1v\ ,\ st_2 \cup t_2v).$$

The invisible region I is called the *obstructed region* due to the line-obstacle O and is denoted by OR (O). We can also partition I into regions I_1 and I_2 so that if $v \in I_1$, PA (s,v) = $st_1 \cup t_1v$ and if $v \in I_2$, PA (s,v) = $st_2 \cup t_2v$. The locus of the partition boundary is given by : $\{x : |t_1x| - |t_2x| = K = |st_2| - |st_1|$, which is a part of a hyperbola and is called the hyp-boundary for the obstacle O. When $K > \emptyset$, this hyp-boundary looks like the broken line (Fig 1) which partitions I into I_1 and I_2.

The shortest-path to any point in I_1 from the source vertex passes through t_1. This point t_1 is called the cusp of the region I_1 and is denoted by CUSP (I_1). Similarly the point t_2 is the cusp of the region I_2 and we can write t_2 =

CUSP (I_2).

Since there are at most 3 sub-regions inside P , any point location problem takes O(1) time. So we can check all the vertices of convex polygon and calculate their shortest-paths according to the regions where they are located in O(n) time. Also , if an arbitrary target point v is given , we can calculate the shortest-path PA (s,v) in O(1) time.

Formally , the above algorithm for partitioning P can be written like this:

Procedure Partition (P : Convex-pol ; s : source-vertex)

{Let t_1 and t_2 be the end points of the
line-obstacle inside P}

begin

Draw tangent lines st_1 and st_2 ;

Put a back pointer to s in each point t_1 and t_2 ;

Extend st_1 and st_2 to meet the boundary of P
at a and b respectively ;

Divide the region $[t_1\ t_2\ b\ a]$ by the locus
of v given by $|t_1v| - |t_2v| = K$ and name the
subregions I_1 and I_2 ;

Let the subregion I_1 gives the locus of the points
whose shortest-path passes through t_1.

$V := P - (I_1 \cup I_2)$;

CUSP $(I_1) := t_1$;

```
    CUSP ( I2) := t2 ;
    CUSP (V) := s ;
  end.
```

The algorithm for finding the shortest-path of a target point from the source point s , after we have obtained the shortest-path partitioning of P for s , is given as:

```
    Procedure find-shortest-path (P :  Convex-pol  ;  s  :
Source-point ; v : target-point )

  Begin
     Locate v inside P and find region R
     in which v is located ;
     Let x := CUSP (R) ;
     PA (s,v) := xv U PA (s,x) , where
     PA (s,x) is obtained following the
     back pointers from x to s ;
  end.
```

ANALYSIS

It is very easy to see that above two algorithms take O(1) time each.

2.2. SIMPLE POLYGON CONTAINING K LINE OBSTACLES

This problem is far more complicated than the previous one. So before we go to the main problem we'll explore the case when polygon P is convex. We use the familiar sweep-

line technique here to partition the polygon P. There are a constant K number of line-obstacles inside the polygon. We can define a type of ordering of these obstacles inside the polygon.

Def: We assign a level number to each of the obstacles. The level of an obstacle can be defined as follows. We define the level of an end point v of an obstacle as one more than the level of the last corner point of the shortest-path from the source point s to v. The level of the source point is taken to be zero. The level of a line obstacle is equal to the maximum of the levels of its end points. So , level of an obstacle visible from the source point s is equal to one.

We can use a sweep-line technique to find out the level of an obstacle. In this technique , a line sweeps from the source vertex towards the obstacles and stops outside the polygon P. In the course of the sweep it calculates the levels of the obstacles it meets inside the polygon. Concurrently , it partitions the polygon P into shortest-path regions. Instead of the sweep-line status , here we use PARTITION_STATUS. At any moment of time the PARTITION_STATUS gives the partitions of the polygon P created till that time. As shown in the last section , for each obstacle O_i , we get an obstructed region OR (O_i). In our algorithm , when the sweep-line meets an line-obstacle , its obstructed region is found out by constructing the tangent lines to its end points. Every end point has two fields as given by the

type declaration below:

```
Type
pointer   = ^end_point
end_point = record
                path   : pointer
                length : integer
            end ;
```

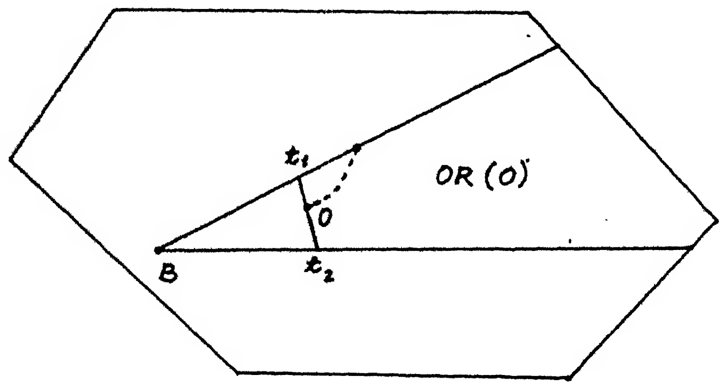
When the sweep-line meets an end point , it sets the pointer field of the end point to point to the origin of the tangent line to this point. Also it finds out the shortest-path length to the origin of the tangent line , calculates the shortest_path length to the end point and stores it in the specified field.

For constructing a tangent line to an end point , the algorithm locates the point in the present partitioning as given by the PARTITION_STATUS and finds out the region it is located. The CUSP of the above region is taken to be the origin of the tangent line to the end point.

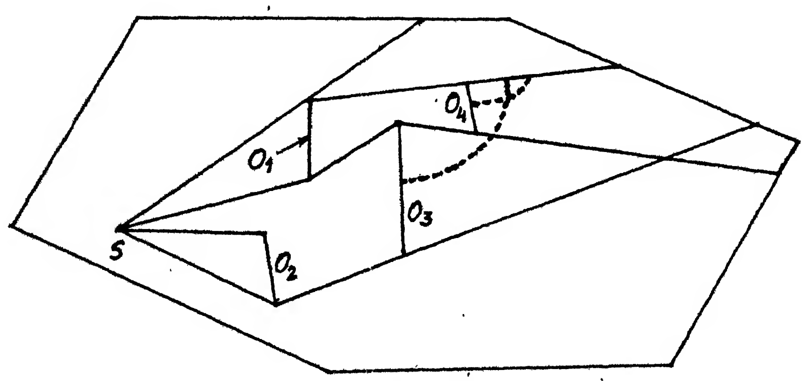
After the obstructed region OR (O_i) for a line-obstacle O_i is found out , we partition the region in two parts by the corresponding hyp-boundary , as discussed in the last section. OR (O_i) is then inserted in the PARTITION_STATUS. Some intersections may be possible between more than one obstructed regions in the PARTITION_STATUS. So , PARTITION_STATUS is to be updated accordingly. Before we

present the formal algorithm , we'll give a list of observa tions , required for the updation of the PARTITION_STATUS :

Obs1: *A hyp-boundary vanishes when it comes out of th corresponding obstructed region.*

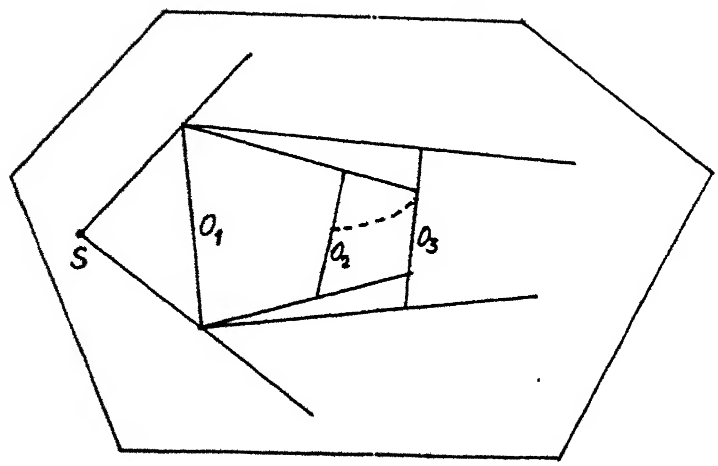


Obs2: *When two obstructed regions intersect , their hyp-boundaries may also intersect with each other. Inter-section of two obstructed regions may require construc-tion of new hyp-boundaries as shown in the following figures*

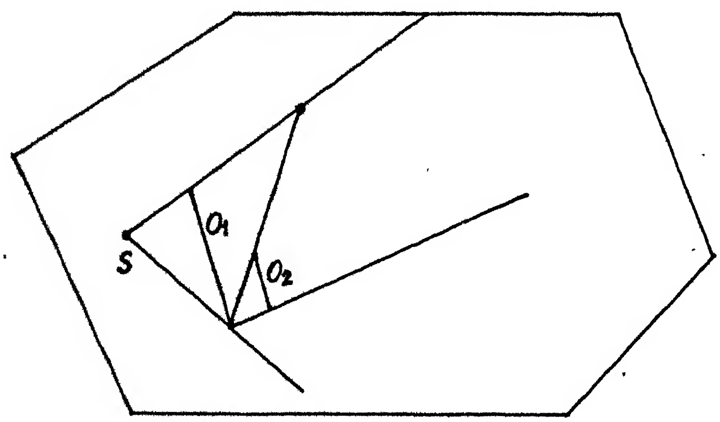


Obs3: *A tangent line as well as a hyp-boundary terminates*

if it meets a line-obstacle as shown in the following figure:



Obs4: *A tangent line terminates when it meets a tangent line of an enclosing obstructed region. An obstructed region is called to be enclosing for a given obstacle O_i if the shortest path to the obstructed region of O_i passes through an end point of the enclosing obstacle (shown in the following figure).*



Formally our algorithm can be given as follows:

```
Procedure Make-partition (P : Convex_pol) ;
 Initialisation:
    Begin
      CHECKED_STACK := φ ;
      PARTITION_STATUS := { The polygon boundary edges which
                             intersect a vertical line through
                             the source vertex } ;
      Sort the vertices of P and the end points
      of the line-obstacles along the abscissa
      and store them in a queue E , the event
      queue for the sweep-line ;
    end ;
  Begin
    Sweep a line from the source vertex s to +(inf.)
    along the abscissa ;
    For each event point e_i perform the following steps:
      STEP1: If e_i is one of the end points
             of a line-obstacle , take the following
             actions:
             STEP1.1: Locate e_i in the PARTITION_STATUS ;
                      Let the region e_i is located be R_i ;
             STEP1.2: p_i := CUSP (R_i) ;
             STEP1.3: Join points p_i and e_i and extend
                      the line.
                      Let the line be denoted by TL (e_i).
                      We call this line as tangent line for the po
```

e_i. Insert the line in the PARTITION_STATUS.Check if the line intersects with any of the polygon boundary edges in the PARTITION_STATUS. Find out the intersections.

STEP1.4: If l be the level of the line-obstacle whose one of the end points is p_i, then assign level $l + 1$ to the point e_i;

STEP1.5: If e_i be an end point of a line-obstacle whose other end point e_j has already been processed and if p_j be the origin of the tangent line TL (e_j), then find the locus l_i of the point x given by the equation:

$|PA(s, e_i)| + |e_i x| = |PA(s, e_j)| + |e_j x|$

This is an equation of a hyperbola and we call this curve l_i as hyp-boundary with focii e_i and e_j.

Extend l_i to meet the bondary of P or TA (e_i) or TA (e_j) after which it vanishes. TA (e_i), TA (e_j) and l_i form two regions. Let the region bounded by TA (e_i) and l_i be denoted by I_i and the other region bounded by TA (e_j) and l_i be denoted by I_j. Then,

$$CUSP(I_i) := e_i$$

$$CUSP(I_j) := e_j$$

Insert the above three regions in the PARTITION_STATUS. Check intersection with the other regions present in PARTIRION_STATUS and accordingly modify PARTITION_STATUS.This modifications should be done in line of the observations listed above.

The level of the obstacle := maximum of the levels of its end points.

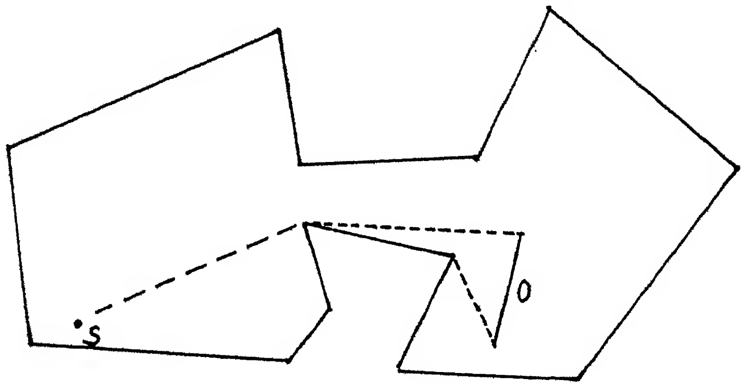
STEP2: If the event point e_i is one of the vertices of the polygon P , then take the following actions:

STEP2.1: Remove the boundary edge of P which ends at the point e_i from the PARTITION_ STATUS. Remove with this edge all the lines intersecting it from the PARTITION_STATUS ;

STEP2.2: Insert the new boundary edge of P which start from the event point e_i in the PARTITIO_ STATUS. Check if any line , currently inside PARTITION_STATUS , intersects this new edge. Find out all the intersections ;

Once the above algorithm is known it is not a major problem to extend it to the case when P is a simple polygon with n vertices. The only problem in the case of a simple polygon is that the polygon boundary also sometimes behave as an obstacle. As seen in the following figure , the obstacle O is not visible from the source vertex. So constructing a tangent line poses some difficulties. We have

to take a polygonal path to reach the end points of the line-obstacle O.

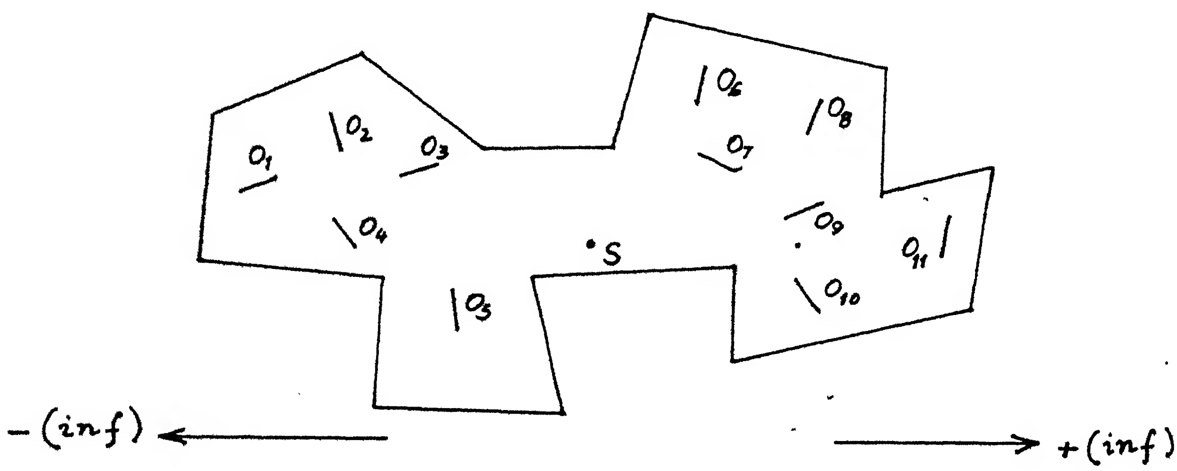


We can circumvent this difficulty by performing a bit of preprocessing. What we do is that before we apply our sweep-line algorithm , we perform a shortest-path partitioning of the simple polygon P , removing all the obstacles from inside. There is already an O(n log log n) algorithm available [2] to do this type of partitioning of a simple polygon without any obstacles inside. We can use this algorithm as a preprocessing algorithm. Once we do this initial partitioning , we can locate any point in P in O(log n) time , find the shortest path , draw the tangent lines and modify the partitioning by the sweep-line strategy.

Once the partitioning is available , finding a shortest path from the given source point to an arbitrary target point becomes an easy job. The same find_shortest_path algorithm , presented in the last section , can also be used here for that purpose.

Before we start analysing the time complexity of the

above algorithm we shall like to mention that for some distributions of obstacles and the source point it may be required that the above algorithm be applied twice. Once while sweeping the polygon from the source vertex to +(inf) and next time sweeping from the source vertex to -(inf). An example of that type of distribution is given below:



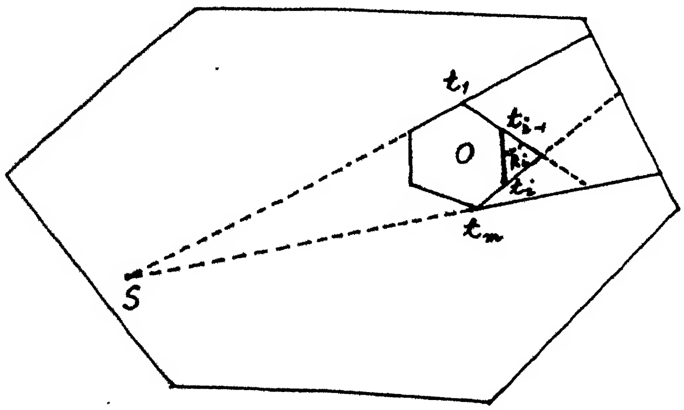
ANALYSIS

It is easy to see that the number of partitions of th polygon P is totally dependent on the constant K and th total number of levels of line-obstacles , which is als constant for a distribution of line-obstacles and a give source point. So , the time complexity of the above sweep- line algorithm is O(1) when P is convex. But when P is simple polygon , we perform an initial preprocessing whic partitions P in O(n) regions. For each end point of th line-obstacles we have to do a point location. Since ther are O(1) line-obstacles inside the polygon , the sweep-lin algorithm takes O(log n) time in the worst case. For conve

polygon , the find_shortest_path algorithm takes O(1) time and for a simple polygon it takes O(log n) time.

2.3. CONVEX POLYGON CONTAINING A SINGLE POLYGONAL OBSTACLE

A typical partitioning of the convex polygon P is given below when there is a single polygonal (convex) obstacle inside it.



The algorithm is easy and straight forward in this case. We don't require any sweep-line strategy to solve this problem. The Steps required to do this partitioning is given below:

STEP1: Draw two tangents from the source to the polygon P. Let the points of contact be t_1 and t_m. Extend the tangent lines to meet the polygon boundary at a and b respectively. The region $[t_1, t_2, \ldots, t_m, b, a]$ is invisible from s and we denote this region by I ;

STEP2: Extend the edges $t_{i-2}t_{i-1}, t_i t_{i+1}, t_i$ ~~$t_1 t_1$~~ t_1 inside

the region I. Let the region formed be called as R_i. For any point in the region R_i , the shortest-path passes through either t_{i-1} or t_i ;

STEP3: Divide each region R_i in two parts by the hyp-boundary l_i with focii t_{i-1} and t_i (as discussed earlier). The hyp-boundary l_i divides the region R_i into regions R_{i1} and R_{i2}.

STEP4: Let the shortest-paths for the points in R_{i1} and R_{i2} pass through t_{i-1} and t_i respectively.

Then ,

CUSP $(R_{i1}) := t_{i-1}$

CUSP $(R_{i2}) := t_i$

Once we perform the partitioning finding a shortest-path from s to an arbitrary target point X is just a point location problem. The same find_shortest_path algorithm can be used here for that purpose.

Once we have an algorithm for a convex polygon we can extend it to the case when P is a simple polygon in the same fashion described in the last section.

3. DISCUSSIONS

We can extend the above algorithm for a single polygonal obstacle to the case when the number of polygonal obsta-

cles is more than one. We can use the same sweep-line technique for this purpose. Since in this case the partitioning of the polygon becomes very complicated , no formal algorithm is given here.

This problem can also be solved by taking each polygonal obstacle as a combination of line-obstacles and then using the same sweep-line algorithm.

It remains an open problem to determine whether such a partitioning is possible for a three dimensional case. Also there are a number of open problems closely related to this shortest-path problem. A problem of major importance is to see whether it is possible to design an algorithm for *geodesic Voronoi diagram* when the simple polygon contains a number of obstacles.

4. REFERENCES

[1] R.E. Tarjan and C. Van Wyk , "An O(n log log n) time algorithm for triangulating a simple polygon" , manuscript , August 1986.

[2] L.Guibas , J. Hershberger , D. Leven , M. Sharir , and R.E. Tarjan , "Linear time algorithms for visibility and shortest path problems inside a simple polygon" , Proc. ACM Symp. on Computational Geometry 1986.

[3] W. Lenhart , R. Pollack , J. Sack , R. Siedel , M. Sharir , S. Suri , G. Toussaint , C. Yap , and S. Whitesides , "Computing the link center of a simple polygon", Proc. 3rd. ACM Symp. on Computational

Geometry.

[4] G. Toussaint , "An optimal algorithm for computing the relative convex hull of a set of points in a polygon" , Proc. of EUSIPCO'86 , Hague , Sep. 1986.

CHAPTER 5

CONCLUSION

In this thesis we have looked at various kinds of *proximity problems.*

In chapter 2 we have outlined an optimal O(n log n) algorithm for the *closest pair* problem for a planar point set , based on the novel concept of a sweep-rectangle. However , this optimality does not carry over to higher dimensions. But this does not detract from the practical usefulness of the algorithm since the number of dimensions is less than three in most practical applications. The sweep-line strategy is primarily suited for problems regarding line segments (for example : line intersection problem). The strategy called sweep-rectangle technique , proposed in this chapter , is a variation of sweep-line and is suitable for problems regarding point sets. It would be interesting to investigate if the sweep-rectangle technique can be successfully applied to other problems regarding points.

In chapter 3 we have discussed a type of generalisation of Voronoi diagram called *Kth degree Voronoi diagram.* In this chapter we have also proposed a new algorithm for dynamic updates of a Kth degree Voronoi diagram. It remains an open problem to determine whether a specialised dynamic

data structure can be designed for the Kth degree Voronoi diagram. There is scope of further work in the following directions: dynamisation of order-K Voronoi diagram ; dynamisation of geodesic Voronoi diagram ; designing an algorithm to construct order-K or Kth degree Voronoi diagram ; designing sweep-line algorithm for geodesic Voronoi diagram ; Kth degree Voronoi diagram , and order-K Voronoi diagram.

In the third chapter we have dealt with a different kind of proximity problem called shortest-path partitioning of a simple polygon. In this chapter we have proposed a sweep-line algorithm to partition a simple polygon containing a number of obstacles. It remains an open problem to determine whether such a partitioning is possible for three dimensional case. An open problem of major importance is to see whether it is possible to design an algorithm for *geodesic Voronoi diagram* when the simple polygon contains a